Technical Report **1757**
September 1997

Network Memory Protocol

D. R. Wilcox

19971203 186

Naval Command, Control and Ocean Surveillance Center
RDT&E Division, San Diego, CA 92152–5001

Technical Report **1757**
September 1997

Network Memory Protocol

D. R. Wilcox

Approved for public release; distribution is unlimited.

Naval Command, Control and Ocean Surveillance Center
RDT&E Division, San Diego, CA 92152–5001

NAVAL COMMAND, CONTROL AND OCEAN SURVEILLANCE CENTER RDT&E DIVISION
San Diego, California 92152–5001

H. A. WILLIAMS, CAPT, USN
Commanding Officer

R. C. KOLB
Executive Director

ADMINISTRATIVE INFORMATION

The work described in this report was performed for Space and Naval Warfare Systems Command by a member of the Systems Branch, Code D4122, within the Advanced Concepts and Systems Technology Division, Code D41, of the Naval Command, Control and Ocean Surveillance Center (NCCOSC) RDT&E Division (NRaD). The work was performed from October 1996 through September 1997 and was funded by the NRaD Internal Research Program under task ZU36.

Released by
M. W. Morgan, Head
Computer Systems Branch

Under authority of
R. B. Volker, Head
Advanced Concepts and
Technology Division

This technology may be the subject of one or more invention disclosures assignable to the U. S. Government. Licensing inquiries may be directed to:

Harvey Fendelman
Legal Counsel for Patents
NCCOSC, Code D0012
San Diego, CA 92152–5765
(619) 553–3001

SB

EXECUTIVE SUMMARY

OBJECTIVE

This report describes research that sought to develop a computer local area network transport layer protocol demonstrating the feasibility of implementing reflective memory concepts on standard local area networks. Unlike the tradition socket interface, the reflective memory approach accesses remote processor memory directly without the overhead of a user server task. The desire to obtain the benefits of reflective memory without reliance on proprietary hardware motivated the work. The research targeted primarily high-speed access to real-time databases such as encountered in tracking, display, and control applications.

METHOD

The new protocol, designated the network memory protocol (NMP), provides network clients with access to memory residing on a network memory host through access request and reply messages. Operating system kernel software executing on the network memory host manages the allocation, address mapping, and access control of the network memory residing on that host. Application tasks executing on the network memory host avoid system calls by mapping the portion of network memory of interest into their user address space. Simple algorithms provide atomic access to data without resorting to system calls.

CONCLUSION

NMP implements local area network access to high-throughput, real-time databases more efficiently than servers employing traditional sockets. The efficiency is gained by avoiding unnecessary context switches and data buffer copies. Testing revealed that NMP access times were approximately two-thirds those required by the User Datagram Protocol (UDP), and three to four times faster than those for the Transmission Control Protocol (TCP). Note, however, that NMP does not provide many of the services provided by UDP and TCP, such as packet routing, reassembly, and error recovery. Since NMP provides random access to data, it is more flexible than socket approaches, which are inherently sequential. The socket approach, on the other hand, does a better job of activating tasks in response to associated message arrivals.

CONTENTS

Figures

1. NETWORK MEMORY MODEL

The **network memory protocol** is a local area network transport layer protocol which provides clients with direct access to data within the random access memories of servers. The server random access memories accessed by the network memory protocol are called **network memories**. A processor implementing a network memory server is called a **network memory host**.

The network memory protocol, which accesses memory through a local area network, is conceptually similar to direct memory access protocols, which access memory through memory hardware interfaces or through processor backplane busses. Although the formats are different, both the network memory protocol and the direct memory access protocols seek the same goal. They seek to improve performance by accessing processor memory asynchronously, and largely independently, of other activity on the processor.

The motivation for the network memory protocol was the desire to build very efficient real-time databases. A **real-time database** is a structured collection of data representing the state of a dynamic external environment. Each database entry contains the sampled value of an associated environmental parameter obtained or derived from an external sensor. The network memory protocol provides a means for remote sensor clients distributed over a local area network to broadcast their sample data directly to the memories implementing the real-time databases. It also provides a means for remote user clients distributed over a local area network to obtain data directly from the memories implementing the real-time databases. These relationships are shown in Figure 1.

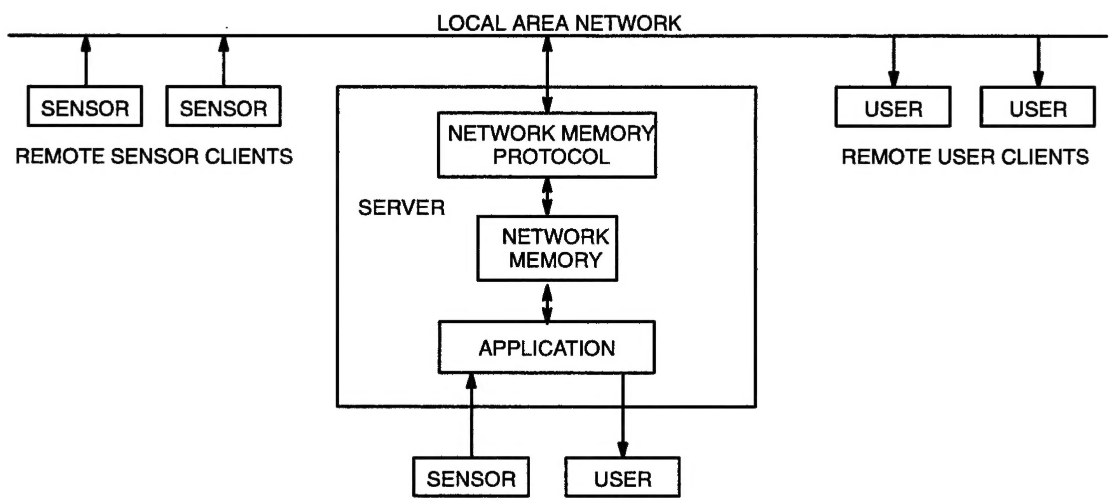

Figure 1. Network memory model.

The operating system of the network memory host manages the network memory. It determines how much space to allocate for network memory, how global network memory protocol addresses are mapped into the allocated space, what categories of network memory protocol accesses to permit, and when to permit them. This enables a network memory to receive the sensor data of interest, when it is of interest, while ignoring the rest.

Application tasks executing on a network memory host access the local network memory by mapping the portion of the kernel address space containing the network memory into their own user

address space. Once the mapping is initialized, the application task can directly access the network memory without the need for additional system calls.

A network memory may contain one or more location monitors. A **location monitor** is a network memory location equipped with hardware or software logic that generates an interrupt within the network memory host when the network memory protocol accesses that memory location. Like other aspects of the network memory, the operating system of the network memory host manages the presence and location of its location monitors. The same global network memory protocol address may point to a location monitor in some processors and to an ordinary location in others.

2. COMPARISON TO THE SOCKET APPROACH

Many of the properties, advantages, and disadvantages of the network memory approach are illustrated by comparison to the traditional socket approach. The network memory approach is not intended to replace the socket approach. It is intended, rather, to provide an alternative for those applications where the network memory approach has an advantage.

2.1 SERVER PROCESSING

In the socket approach, each processor containing a real-time database executes a server task to accept data from remote sensor clients. When one or more messages from the sensors become available, the processor operating system kernel activates the server task. The server task uses a socket read, or similar system call, to copy a message from the network buffer in the kernel address space into a buffer in its own user address space. It then decodes the message to determine the type of data it contains. Finally, it copies the data from its user buffer to the appropriate entry in the real-time database. The server task repeats the process while additional messages remain available.

The network memory approach, on the other hand, does not require the intervention of a separate server task scheduled by the kernel. The kernel copies the sensor data from the network buffer directly into the real-time database entry specified by fields within the message itself.

The socket approach suffers from two inefficiencies in comparison to the network memory approach.

First, the socket approach socket read or similar system call requires a context switch from the server task to the kernel, and when messages become available, another context switch back again to the server task. The network memory approach does not require these context switches. It executes everything within the kernel. It is, therefore, more efficient.

Second, the socket approach copies the sensor data twice. The first copy is from the kernel network buffer to a user buffer in the server address space. The second copy is from this user buffer to its final destination at the real-time database entry. The server task does not have direct access to the message content in the kernel address space. The server task needs direct access to the message content to determine the desired destination database entry. The server, therefore, must first request the kernel to copy the data into a buffer in the server address space.

The network memory protocol, on the other hand, requires only one copy. Since it executes in the kernel address space, it has direct access to the network memory buffer in the kernel address space. It copies the sensor data from the kernel network buffer directly to the database entry. The network memory approach is more efficient because it does not require an intermediate copy of the message content.

2.2 ACCESS SEQUENCE

The socket approach presents received messages sequentially. A server receiving multiple message types through the same socket must decode the content of each message to determine the appropriate processing. Since the server does not know the message type until it decodes the message content, it must examine all messages received through the socket to obtain the messages of current interest.

Applications have random access to the data in the network memory. The kernel places the data of each message at the memory address specified by the message itself. Although the kernel is itself decoding received messages sequentially, it is doing so asynchronously to user applications and much more efficiently than could be done by a user server task. Applications can access data of current interest without the burden of decoding all messages received. They can also access the highest priority data first rather than in the sequence of arrival.

The socket approach can prioritize messages. One way to do this is for the kernel to place the higher priority messages at the front of socket input queue so that the server receives them first. While this may help, it is not very flexible. The number of defined socket message priority levels is usually limited to two, a normal priority level and a higher "out-of-bound" priority level. Furthermore, the message priority is determined by the client rather than by the server. Different servers listening to the same port cannot independently adjust the priority of message types to meet their own needs. They are all limited to the priority scheme established by the client.

A better solution is to assign separate port numbers for each message type. The kernel sorts the received messages by port number without regard for priority. Servers create a separate socket to receive messages from each port of interest. Each server can access the messages it considers the highest priority first by reading from their respective sockets first.

The network memory approach permits priorities to be assigned to message data locally because the data can be accessed from the network memory in any desired order. The system can also change the priority scheme as system conditions change without notifying the clients.

2.3 TASK ACTIVATION

The network memory approach is ideally suited to the real-time database applications that motivated it. It gains efficiency by performing the required functions directly within the kernel without the overhead of a server task executing in user space. But there are also applications that require received data to be processed in some manner immediately upon arrival rather than simply stored in a database.

The network memory protocol does not explicitly define a means of activating a server task to perform such processing upon message arrival. The network memory protocol can indirectly activate a server task, however, by accessing a network memory location monitor. The location monitor interrupt activates the server task.

The socket approach associates each socket with a list of server tasks waiting for messages from the respective socket. The kernel automatically places the server tasks into the ready state when the message arrives. The socket approach, therefore, is well-suited for applications requiring immediate processing of messages upon their arrival.

If location monitors are available where needed, the network memory approach can be more flexible than the socket approach. Consider a system, for example, where a sensor client periodically broadcasts data samples indicating the position of a physical object to two processors. One processor supports a high-resolution tracker application that accurately predicts the future position of the object based upon all the received sample positions from the sensor. The other processor supports a display which, due to the display resolution, can skip sensor position samples as long as the period between the samples it accepts is not too great.

Using the socket approach, each processor contains a server that accepts all sensor position sample messages. This is fine for the processor supporting the high-resolution tracker application since it needs to receive all the samples to predict the future position of the object accurately. The processor supporting the low-resolution display application does not need to receive every sample. It would prefer to skip the processing of unnecessary samples in order to conserve its processing resources for other applications. Unfortunately, the socket approach forces it to receive every sample, even if the sample is not needed.

The network memory approach with location monitors provides a better solution. The processor supporting the high-resolution tracker application uses a location monitor to activate the application task when each sensor sample data message arrives. The operating system of the processor supporting the low-resolution display application activates its display application task periodically at a rate consistent with the display resolution requirements. The display application task simply reads the latest sample stored in the network memory when it needs it. The sample in the network memory may have been updated many times between successive reads by the display application task. Thus, the task skips the samples it does not need.

3. NETWORK MEMORY DESIGN

Before introducing the network memory protocol message format, it is helpful to examine some of the issues that guided its design.

3.1 BYTE ORDER

The network memory protocol accesses network memory data using the big endian format. The **big endian** format represents the numbers that span more than a single byte such that the most-significant bit of the number is in the byte with the lowest address.

Network memory host application tasks use the `ntohs` and `ntohl` macros to convert network memory 2-byte and 4-byte numeric data, respectively, into host format. They use the `htons` and `htonl` macros to convert host 2-byte and 4-byte numeric data, respectively, into network memory format. Although not required by application tasks executing on big endian network memory hosts, since no conversion takes place, it is good practice to include these macros in all application task source code for clarity and portability to little endian hosts.

The network memory protocol uses the big endian format because it matches the network byte order. This enables all data types to be treated simply as byte strings. There is no need for the network memory protocol to define separate instructions for the various multiple-byte numeric data types. Also, since there is no conversion between network byte order and network memory byte order, any received network memory data that are not used by network memory host application tasks are not needlessly converted.

3.2 NETWORK MEMORY PARTITIONS

Network memory is implemented within the kernel memory of the network memory host. Since the physical memory available to the kernel is generally considered a precious resource, the space allocated to network memory should be no larger than necessary.

A simple and efficient way for the host to map global network memory addresses into kernel memory addresses is to add a differential constant to the address. When all the global network memory addresses of interest at a particular network memory host are clustered within a relatively small range of values, the network memory host can perform this mapping with little or no wasted kernel address space. The system designer should strive to cluster the assignment of global network memory addresses associated with each network memory host to take advantage of this property.

Unfortunately, clustering all the global network memory addresses of interest at a particular network memory host may not be possible. Consider, for example, a system containing three network memory hosts and three clusters of global network memory addresses. Denote the network memory hosts as 1, 2, and 3, and the clusters as A, B, and C. Assume further that the network memory in host 1 maintains copies of clusters A and B, that the network memory in host 2 maintains copies of clusters B and C, and that the network memory in host 3 maintains copies of clusters A and C. If for each host, the two clusters maintained by that host are mapped together into a single contiguous section of kernel address space, then at least one of the hosts wastes kernel address space. Figure 2 illustrates that the amount of wasted space in that the host is equivalent to the size of the cluster not included.

The problem is solved by the host dividing the global network memory address space into independently mapped partitions. Each address cluster of interest is assigned to a separate partition. When the

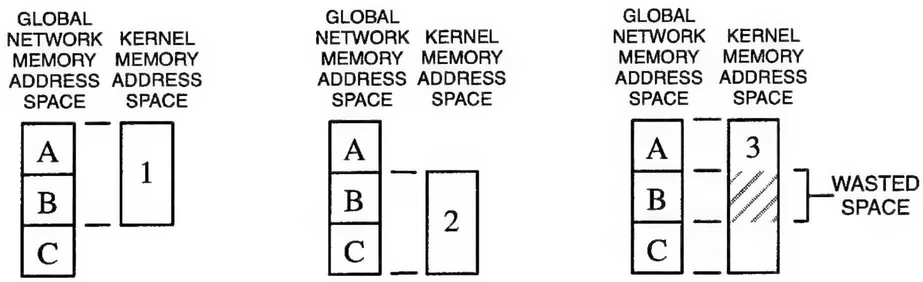

Figure 2. Network memory mapping example.

host receives a client message, it checks the minimum and maximum global network memory address of each partition of interest to determine whether the specified access is within the range of the respective partition. The approach avoids wasting kernel address space because the host only needs to allocates space for those partitions associated with address clusters of interest.

The preceding example used partitioning to eliminate a gap in the contiguous section of kernel memory address space allocated to network memory. One can also use partitioning to split the mapping of a contiguous network memory address space into separate sections of kernel memory address space. In other words, one can also use partitioning to intentionally insert gaps into the kernel memory address space allocated to network memory.

3.3 MEMORY-MAPPED INPUT/OUTPUT

Many computers contain a backplane bus that connects the processing hardware to the input/output hardware through a memory-mapped interface. The processor software controls the input/output hardware by accessing dedicated memory addresses recognized by the input/output hardware. The kernel configuration reserves a separate section of the kernel memory address space for these special addresses.

The network memory protocol does not distinguish between accesses to conventional memory addresses versus those to memory-mapped input/output hardware devices. This enables processors on the local area network to share input/output devices. As is the case for any system where multiple processors share a common resource, proper consideration must be given to coordination.

The kernel memory address space section configured for input/output hardware is not necessarily contiguous with the section used for network memory implemented from conventional memory. Network memory partitions allow the sections to be mapped independently. The same concept is applicable to sections of other types of memory as well, such as read-only memory and non-volatile memory, which may reside in isolated sections of the kernel memory address space.

3.4 NETWORK MEMORY OVERLAYS

Consider a system with several network memory hosts. All the hosts are programmed to respond to the same range of global network memory addresses. When a sensor client writes to a global network memory address within this range, all the hosts write the data into their respective network memories simultaneously. When a user client reads from a global network memory address within this range, all the hosts send a reply message containing the data read from their respective network memories. If there are many hosts, then there will be many reply messages. Usually one would rather have the user

client receive a reply from a single host while still having the sensor client update all the hosts. This can be accomplished using network memory overlays.

A **network memory overlay** is a mapping of network memory partitions such that different partitions of the global network memory address space map into the same section of host kernel memory address space. In other words, each location in the network memory overlay has two or more global network memory addresses for the same location.

The problem presented by the example is solved by using one global network memory address partition for sensor client writes to all hosts and a different one for user client reads at one of the hosts. The partitions are programmed to overlay the same locations in the kernel memory address space. The same data written to all hosts at an address in one partition can then be read back from a single host at another corresponding address in the other partition.

4. UNIX NETWORK MEMORY HOST INTERFACE

This section describes how to allocate and map network memory within a Unix host that has been configured with a network memory driver.

4.1 DEFINING KERNEL MEMORY SPACE FOR NETWORK MEMORY

Unix hosts implement network memory through a character device driver. The network memory driver is similar to those for the `/dev/mem` and `/dev/kmem` nodes. Application tasks locate the network memory driver through one or more `/dev/netmem` nodes. The system administrator uses the Unix `mknod` command to define a separate `/dev/netmem` node for each contiguous section of kernel memory to be made available for network memory.

Every Unix device has a **device number**. The device number bits are divided into two groups called the **major device number** and the **minor device number**, respectively. The system administrator specifies the major and minor device numbers as parameters to the Unix `mknod` command. The kernel uses the major device number to identify the driver desired. The interpretation of the minor device number depends upon the driver implementation.

For character devices, the minor device number typically selects the particular device or interface controlled by the driver. This enables the same driver software to be applied to a set of similar devices. The minor device number can also be used to specify the way the driver controls the device. Magnetic tape drivers, for example, usually use a bit in the minor device number to select whether to rewind the tape or not when the device is closed.

The kernel passes the entire device number to the driver routines as a single parameter. If one defines the least-significant bit position as bit position 0, the major device number usually occupies device number bit positions 15 through 8. The remainder of the bits, both most-significant and least-significant, define the minor device number.

The network memory driver divides the minor device number into a network memory number in the least-significant bits and a network memory size in the remaining most-significant bits. The network memory size is specified as the number of memory allocation units. A memory allocation unit typically contains 4096 bytes.

Figure 3 shows the network memory device number layout. The four least-significant bits specify the network memory number. The network memory size is split between two fields. The least-significant bits are in device number bit positions 7 through 4 and the most-significant bits are in device number bit positions 31 through 16. Other layouts are possible by modifying the macros defining the layout before compiling the network memory driver into the kernel.

The rationale for the layout in figure 3 is as follows. The memory allocation unit is 4096 bytes. It takes 12 bits to represent an address within a given memory allocation unit. Unix device numbers are defined as being type `int` in the language C. Type `int` contains 32 bits in a 32-bit processor. Thus, the number of bits required to address uniquely each allocation unit is $32 - 12 = 20$ bits. The maximum network memory size is one that includes the entire memory address space. Thus, the maximum network memory size can be defined by 20 bits. These 20 bits are split into a group of 16 and a group of 4, as shown in figure 3. Since the major device number is 8 bits, this leaves $32 - 20 - 8 = 4$ bits for the network memory number.

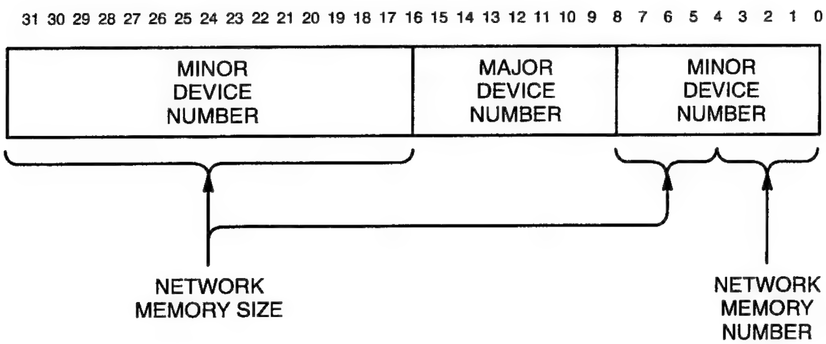

Figure 3. Network memory device number layout.

Assuming the layout design approach just described and an 8-bit major device number field, the number of bits in the network memory number field is related to the size of an allocation unit. If the allocation unit contains 2048 bytes, the network memory number field contains 3 bits. If the allocation unit contains 8192 bytes, the network memory number field contains 5 bits. The number of bits defined by type `int` is not a factor.

4.2 ALLOCATING NETWORK MEMORY

The system administrator uses the Unix `mknod` command to create a `/dev/netmem` node. The `/dev/netmem` node defines the size of a possible network memory. Kernel memory is not actually allocated to the network memory until the application user executes an `open` system call with the `/dev/netmem` node as a parameter. The `open` system call decodes the minor device number to determine the amount of kernel memory to allocate.

Multiple tasks can open the same network memory simultaneously. The first `open` system call for a particular network memory allocates kernel memory to that network memory. Later `open` system calls for the same network memory do not allocate additional kernel memory because it is already allocated.

The kernel keeps track of the number of successful `open` and `close` system calls associated with each network memory. The kernel informs the network memory driver when all the previous opens become closed. The network memory driver frees the kernel memory associated with the network memory on the final close.

4.3 CREATING NETWORK MEMORY PARTITIONS AND OVERLAYS

Network memory partitions and overlays are created using the `ioctl` system call. The `ioctl` parameter has the following C language format:

```
typedef struct {
      NETMEM_ADDR net_addr_min;
      NETMEM_ADDR net_addr_max;
      caddr_t local_addr_base;
      unsigned short access_flags
      unsigned char read_key_length
      unsigned char write_key_length
```

```
        NETMEM_KEY read_key;
        NETMEM_KEY write_key;
    } NETMEM_PARTITION;
```

The `net_addr_min` and `net_addr_max` elements specify the minimum and maximum global network memory byte addresses accepted by the partition. The `local_addr_base` element specifies the base of the partition relative to the base of the section of contiguous kernel memory allocated for the network memory. Defining the same `local_addr_base` element content for two or more partitions creates a network memory overlay.

The `access_flags` element bits inhibit respectively read, write, and pointer access.

The `read_key` and `write_key` elements indicate the optional binary passwords that network memory protocol messages must provide before respectively reading from or writing to the partition. The `read_key_length` and `write_key_length` elements specify the length of the read key and write key. A key length of zero disables the respective key.

The network memory driver maintains a list of the defined network memory partitions for each network memory. It makes the list available to the software implementing the network memory protocol. The network memory protocol only accesses the network memory when it finds an entry in the list giving it an address mapping and permission for the requested access. The network memory protocol, therefore, does not see a network memory that has been opened but has not yet received any partition definitions.

Network memory partitions cannot be created until a network memory is created using the `open` system call. In the case of memory-mapped input/output hardware, however, no kernel memory is allocated. The maximum kernel memory size specified by the device number is, therefore, zero.

4.4 MAPPING NETWORK MEMORY INTO USER SPACE

The application task maps the section of contiguous kernel memory implementing the network memory into application user space using the `mmap` system call. The starting address is relative to the base of the kernel memory section dedicated to the network memory. Once the `mmap` system call has been executed successfully, the application task can access the network memory directly without the need of further system calls.

The mapping between the network memory and the application task user address space is completely independent of the mapping between the network memory and the global network memory address space.

5. COORDINATION ALGORITHMS

One of the goals of the network memory protocol is to provide a set of simple instructions that can be selected and arranged within messages to implement various memory access algorithms. This gives the system designer the flexibility to choose the algorithms that best suit the overall application. The algorithms presented here are intended to show that problems such as coordination can be solved, and to suggest a rationale for the inclusion of various instructions within the protocol.

Both the operating system and the application tasks of a network memory host potentially access the network memory. The operating system accesses the network memory to transfer data between the network memory and the local area network as directed by the network memory protocol. Its accesses are associated with an interrupt from the local area network interface hardware. These interrupts can happen at any time. Thus, the operating system can interrupt an application task while the application task is also accessing the network memory. This can create a coordination problem when the operating system and the application task attempt to access the same set of software-related network memory locations simultaneously and at least one of them is modifying the locations. Although generally rare, such situations are possible and must be addressed.

The application tasks could simply disable network interrupts while accessing the network memory. This would require two system calls, one to disable network interrupts before the access and the other to enable them after the access. This is a poor solution because it degrades performance to protect against a situation that rarely occurs. Fortunately, there are simple alternatives that do not require the costly overhead of system calls.

5.1 SENSOR CLIENT ACCESS COORDINATION

Consider a system in which sensors send data records over a local area network using the network memory protocol. A processor interested in these data opens a network memory to receive the sensor data records. Application tasks executing on the processor access the sensor data records by mapping the network memory into their user address space and reading the data directly from that address space. A situation could arise where an application task begins reading a sensor data record, the operating system interrupts the application task and updates the same sensor data record with newly received sensor data, and then the application task finishes reading the sensor data record. The application task thus obtains a portion of the old record mixed with a portion of the updated record. Since the resulting hybrid is likely erroneous, application tasks often need a means of ensuring that the sensor data records they read from the local network memory are atomic.

The problem can be solved by including a sequence number in the network memory that is associated with the the sensor data record. When the sensor client has new data to send, it constructs a message using the network memory protocol and transmits the message over the local area network. The message instructs any listening network memory hosts to write the new data into their network memory and to increment the associated sequence number.

When the network memory host receives the message, its operating system interrupts lower priority processing to execute the message instructions. This lower priority processing may include an application task that is currently accessing the same sensor data record. The application task first reads the sequence number, then reads the sensor data, and finally reads the sequence number again. If the sequence number contains the same value both before and after reading the sensor data, then the application task knows that the data record was not overwritten by the operating system. If the

sequence number changed, then the application task reads the sensor data and the sequence number again. This process continues until the last two reads of the sequence number yield the same value. Figure 4 shows a flowchart of the algorithm.

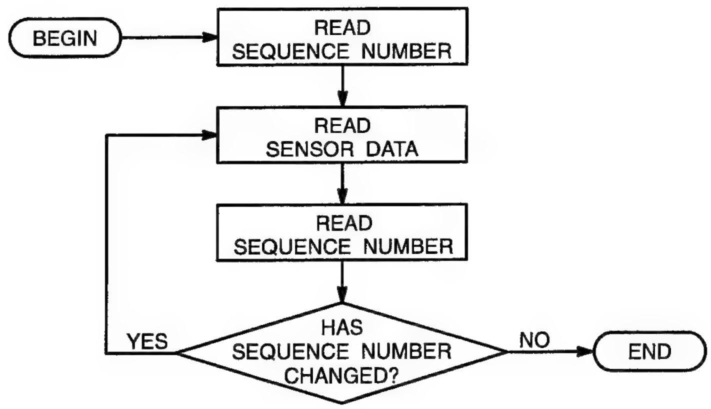

Figure 4. Local atomic read flowchart.

The network memory, rather than the sensor client, maintains the value of the sequence number. In addition to functions such as read and write, the network memory protocol provides a function to increment the content of an address. The sensor client uses this increment function to increment the sequence number stored in the network memory. The sensor client does not need to know the current value of the sequence number in order to increment it. This enables the algorithm to work with more than one sensor client updating the same sensor data record since each sensor client does not need knowledge of the activity of the other sensor clients.

Although the operating system may interrupt the application task to update the sensor data record, the application task will never interrupt the operating system while the operating system is updating the sensor data record. The entire transaction performed by the message will appear to the application task as atomic. It does not matter, therefore, where the sensor client places the instruction to increment the sequence number relative to the other instructions within the message.

The sequence number also provides a simple way for an application task to detect whether it has missed any sensor data record updates since it last accessed the sensor data record and, if so, how many sensor data record updates were missed.

For systems where both the sensor data messages are processed immediately upon arrival and where the period between successive sensor data messages is relatively long, the sequence number and associated algorithm are unnecessary. In such systems, the network memory protocol executing within the operating system does not overwrite old sensor data with new sensor data until long after all the application tasks have finished accessing the old sensor data. It is recommended, however, that sensor clients support the sequence number to provide the needed flexibility in case this situation changes in the future.

5.2 USER CLIENT ACCESS COORDINATION

Consider a system in which remote clients use the network memory protocol to obtain data records over a local area network from a network memory host. Application tasks executing on the network

memory host update the data records. They map the network memory into their user address space and write directly into that address space. A situation could arise where an application task executing on the network memory host begins updating a data record, the operating system interrupts the application task to read the same data record on behalf of a remote client, and then the application task completes updating the data record. The remote client would then obtain a portion of the old record mixed with a portion of the updated record. Since the resulting hybrid is likely erroneous, remote clients often need a means of ensuring that the data records obtained from the remote network memory are atomic.

One solution is to include a flag in the network memory that is associated with the data record. The flag indicates whether the application task is currently updating the data record. The application task sets the flag before beginning the update and clears the flag after completing the update. If the remote client sees the flag as cleared, it knows that the received data record is atomic; if it sees it as set, it knows that it is not atomic. Upon detecting reception of a non-atomic data record, the client simply repeats the data record request until receiving an atomic version of the data record. Figure 5 shows a flowchart of the algorithm.

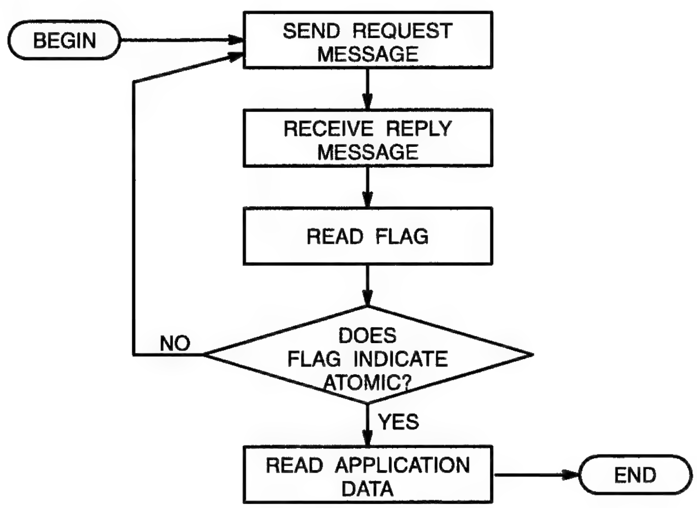

Figure 5. Remote atomic read flowchart.

While seemingly simple, this approach has a number of disadvantages. An obvious disadvantage is that repeated requests for the data record increase local area network traffic. Furthermore, transmitting the request and receiving the reply for each attempt takes execution time and adds complexity within the client. Another problem is that the client has no visibility into how long the network memory host application task will need to complete its data record update. The client, therefore, cannot determine the optimum time to wait before repeating its data record request.

Another solution is for the clients to access the network memory data records indirectly through pointers also in network memory. When a network memory host application task updates a network memory data record, it first makes a copy of the data record that also resides in network memory. The pointer used by the client remains pointing to the original data record while the application task updates the copy. Client read accesses to the original data record remain atomic since the original data record is not altered. When the application task completes its update of the copy, it modifies the pointer so that it now points to the copy. The copy thus becomes the new original. The old original is freed to provide space for the copy of a future data record update.

The first solution, using the flag, is very efficient as long as the network memory host application task data record updates do not conflict with remote client reads of the same data records. Latency degrades significantly when a conflict, although hopefully rare, does occur. The degradation is due primarily to the need to process additional network memory protocol messages. In the absence of conflicts, the second solution is less efficient than the first because it copies the data record. This may become an issue for large data records. The second solution has the advantage that conflicts, when they do occur, do not degrade latency.

5.3 TIME SYNCHRONIZATION

Consider a real-time system in which a network memory host receives sample values of a parameter from a client sensor using the network memory protocol over a local area network. A network memory host application task uses the received parameter sample values to predict the value of the parameter for the time between samples. The application task requires not only the sample values, but also the time each respective sample value was captured, to calculate the prediction.

Ideally, the sample value time should be captured at the same instant as the respective sample value. The local area network and the software activity of the network memory host introduce delay from the time the sensor client captures the sample to the time that the application task receives the sample. This delay varies in duration from sample to sample. Therefore, the best real-time clock to capture sample time is one at the sensor client. When this is not practical, the next best choice is a real-time clock at the network memory host local area network interface.

The network memory protocol provides an instruction which, when executed, stores the current value of the network memory host real-time clock into the network memory. Sensor clients may time-stamp their data samples with the time they arrive at the network memory host by including this instruction within their network memory protocol sample data message. Since there are usually variations in the time between when the sample arrives at the network memory host to when the application task has an opportunity to process it, the network memory protocol provides a more accurate time stamp than one taken within the application task.

The network memory protocol time-stamp instruction also supports algorithms that synchronize the real-time clocks of the network memory hosts to each other. A client, normally one implemented within one of the network memory hosts, periodically broadcasts a network memory protocol message containing the time stamp instruction to all the network memory hosts on the local area network. This message acts as a strobe that captures the state of all the respective real-time clocks at a given instant in time. The local real-time clock value is adjusted by comparing it against the values at other hosts that were captured by the same strobe. The real-time clock rate is adjusted by comparing the time value precession from one strobe to the next.

6. NETWORK MEMORY PROTOCOL SERVER INSTRUCTIONS

All network memory protocol messages are composed of network memory protocol instructions. The message instructions form a small program defining the network memory access. The instructions are similar to those of a very simple hardware processor. This section describes each network memory protocol instruction in detail.

6.1 ADDRESS REFERENCE REGISTER

The software that implements the network memory protocol at the network memory host contains a variable called the **address reference register**. The address reference register fully defines the address of a byte in the network memory global address space. The beginning of each network memory protocol message should include instructions to load the address reference register. All the instructions that access network memory data compute the address to access relative to the content of the address reference register.

Global network memory addresses generally require a large number of bits. As presented previously, the address assignments for locations of interest within a network memory can often be clustered within a limited range of addresses. The purpose of the address reference register is to specify the base address of a limited range of addresses. The instruction which actually access the network memory can then use a much smaller offset field to address locations within that range relative to the base address. This is advantageous for a message containing a sequence of instructions that access locations within the same limited range because the many bits defining the base address only need to be coded once for the entire sequence.

6.2 INSTRUCTION FORMAT

Figure 6 shows the network memory protocol instruction format.

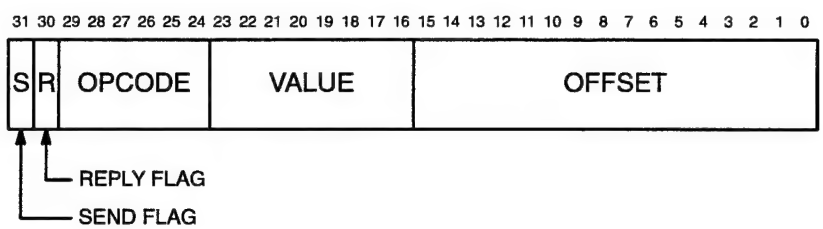

Figure 6. Network memory protocol instruction format.

All network memory protocol instructions are 4 bytes long. They are all aligned on a 4-byte boundary within the data field of the local area network packet. Some instructions have appended byte strings. The byte strings are also aligned on 4-byte boundaries. They are padded, if necessary, to fill the gap to the next four-byte boundary. Instruction fields specifying byte string length do not include the padding.

The network memory protocol server software works with linked lists of small buffers containing the raw data received from the local area network. In the Berkeley UNIX world, these buffers are called **mbuf**'s. In the LINUX world they are called **skbuf**'s. Although these buffers come is various forms that support various maximum raw data lengths, they all have the common property that the maximum raw data length is a multiple of 4 bytes. Making all network memory protocol instructions

4 bytes long greatly simplifies the implementation because instructions are never split across the boundaries of these small buffers.

The value and offset fields are aligned on byte boundaries so that network memory protocol software can use byte-oriented processor instructions. This avoids the use of more complex processor instructions, or sequences of instructions required to manipulate unaligned bit fields.

6.3 SEND AND REPLY FLAGS

The instruction most-significant bit positions 31 and 30 contain the send flag and reply flag, respectively. The **send flag** informs the network memory server that the client wants the instruction returned as a reply instruction. The **reply flag** indicates that the instruction is a reply instruction.

If the send flag is set by the client, the responding network memory server sets the reply flag before returning the instruction. If no errors occurred during processing of the instruction, the server also clears the send flag. Network memory servers ignore instructions with set reply flags because they are directed to a client.

When a client transmits a read instruction to the server, it must set the read instruction send flag to get back the value read. If the client does not need the read data, but is only interested in using the read instruction to trigger a location monitor, then it can leave the send flag cleared.

Clients normally do not set the send flag when they transmit write instructions to servers since the write instruction does not return new user data. The write instruction send flag can be useful, however, for providing the client with verification that the write data returned matches what was sent and that no errors occurred during instruction execution.

The reply flag may seem unnecessary since, by definition, anything that a client receives back from a server should be a reply. The problem arises when the same processor acts as both a client and as a server. Consider the following example. A local processor, acting as a client, transmits a message requesting a remote processor to read some data and return the results back to the local processor. Shortly thereafter, the local processor receives a message back from the remote processor. If the message received is a reply to its original read request, then the local processor should transfer the read data from the message into its network memory. Suppose, instead, that the received message is a read request from the remote processor also acting as a client. Then the local processor, acting as a server, should transfer the read data in the other direction, from its network memory to the message. Since the functions differ, the local processor needs the reply flag to distinguish between a reply from a remote server and a new server request from a remote client.

The reply flag also prevents the development of protocol infinite loops. Consider the consequences of broadcasting a request message that generates a reply message. If there is no way to distinguish between the request message and the reply message, the reply message might be misinterpreted as another request message that would also generate a reply. This could happen at every processor that receives the broadcast. The resulting chain reaction could saturate the network bandwidth with useless messages. The reply flag prevents this from happening because a reply message never generates another reply message.

6.4 SERVER INSTRUCTION OPCODES

The instruction opcode field specifies the function performed by the instruction. The opcodes are listed below. The number to the left of each entry is the opcode value in hexadecimal.

00 Return Instructions

The **Return** instruction marks the end of a network memory protocol message. It is included because some local area networks require packets to have at least a minimum number of data bytes and that minimum may be larger than required by a network memory message. Ethernet, for example, requires packets to contain at least 46 data bytes.

01 Load Address Instruction

The **Load Address** instruction replaces the contents of the three least-significant bytes of the address reference register with the contents of the three least-significant bytes of the instruction. The remaining bytes of the address reference register are unchanged.

02 Load Address String Instruction

The **Load Address String** instruction loads the contents of the bytes immediately following the instruction value field into the address reference register. The instruction value field specifies the number of bytes to be loaded. If the length of the byte string minus two is both positive and not a multiple of 4 bytes, additional bytes are appended to fill the space to the next 4-byte boundary. The additional bytes are not included in the length specified by the instruction value field.

If the address byte length is less than or equal to three, it is recommended that the Load Address instruction be used, rather than the Load Address String instruction, because the former is likely to execute faster within the server.

03 Load Pointer Instruction

The **Load Pointer** instruction loads the address reference register with a pointer read from the network memory. The pointer is located at the address specified by the address reference register plus the unsigned contents of the instruction offset field. The number of bytes forming the pointer stored in the network memory is host-dependent, but shall not exceed 8 bytes. When the number of bytes forming the pointer is less than the number of bytes in the address reference register, the pointer replaces the respective least-significant bytes of the address reference register and leaves the most-significant bytes unchanged.

This instruction enables the network memory protocol to access data at locations selected by a network memory host application task.

04 Load Read Key Instruction

The **Load Read Key** instruction loads the contents of the bytes immediately following the instruction value field into the read key register. The instruction value field specifies the number of bytes to be loaded. If the length of the byte string minus two is both positive and not a multiple of 4 bytes, additional bytes are appended to fill the space to the next 4-byte boundary. The additional bytes are not included in the length specified by the instruction value field.

05 Load Write Key Instruction

The **Load Write Key** instruction loads the contents of the bytes immediately following the instruction value field into the write key register. The instruction value field specifies the number of bytes to be loaded. If the length of the byte string minus two is both positive and not a multiple of 4 bytes, additional bytes are appended to fill the space to the next 4-byte boundary. The additional bytes are not included in the length specified by the instruction value field.

06 Read Byte Instruction

The **Read Byte** instruction reads the contents of the network memory byte at the address specified by the contents of the address reference register plus the unsigned contents of the instruction offset field. The read byte is inserted into the instruction value field.

The instruction generates an error, rather than performing the read, if the read key associated with the addressed partition is active and does not match the read key register.

07 Write Byte Instruction

The **Write Byte** instruction writes the contents of the instruction value field into the network memory at the address specified by the contents of the address reference register plus the unsigned contents of the instruction offset field.

The instruction generates an error, rather than performing the write, if the write key associated with the addressed partition is active and does not match the write key register.

08 AND Byte Instruction

The **AND Byte** instruction replaces each bit of a network memory byte with a bit that is the logical AND of the original bit and the bit in the respective bit position of the instruction value field. The network memory byte is located at the address specified by the contents of the address reference register plus the unsigned contents of the instruction offset field.

The instruction generates an error, rather than modifying the byte, if the write key associated with the addressed partition is active and does not match the write key register.

09 OR Byte Instruction

The **OR Byte** instruction replaces each bit of a network memory byte with a bit that is the logical OR of the original bit and the bit in the respective bit position of the instruction value field. The network memory byte is located at the address specified by the contents of the address reference register plus the unsigned contents of the instruction offset field.

The instruction generates an error, rather than modifying the byte, if the write key associated with the addressed partition is active and does not match the write key register.

0A Exclusive-OR Byte Instruction

The **Exclusive-OR Byte** instruction replaces each bit of a network memory byte with a bit which is the logical exclusive-OR of the original bit and the bit in the respective bit position of the instruction value field. The network memory byte is located at the address specified by the contents of the address reference register plus the unsigned contents of the instruction offset field.

The instruction generates an error, rather than modifying the byte, if the write key associated with the addressed partition is active and does not match the write key register.

0B Add Byte Instruction

The **Add Byte** instruction replaces the contents of a network memory byte with the arithmetic sum of the contents of the original byte and the contents of instruction value field. The network memory byte is located at the address specified by the contents of the address reference register plus the unsigned contents of the instruction offset field.

The instruction generates an error, rather than modifying the byte, if the write key associated with the addressed partition is active and does not match the write key register.

0C Read Byte String Instruction

The **Read Byte String** instruction reads the contents of the byte string from network memory starting at the address specified by the contents of the address reference register plus the unsigned contents of the instruction offset field into the message bytes that immediately follow the instruction. The length of the byte string is specified by the instruction value field. If the length of the byte string is not a multiple of 4 bytes, additional bytes are appended to fill the space to the next 4-byte boundary. The additional bytes are not included in the length specified by the instruction value field.

The instruction generates an error, rather than performing the read, if the specified string extends beyond the limit of the addressed partition, or if the read key associated with the addressed partition is active and does not match the read key register.

0D Write Byte String Instruction

The **Write Byte String** instruction writes the contents of the byte string that immediately follows the instruction into the network memory starting at the address specified by the contents of the address reference register plus the unsigned contents of the instruction offset field. The length of the byte string is specified by the instruction value field. If the length of the byte string is not a multiple of 4 bytes, additional bytes are appended to fill the space to the next 4-byte boundary. The additional bytes are not included in the length specified by the instruction value field.

The instruction generates an error, rather than performing the write, if the specified string extends beyond the limit of the addressed partition, or if the write key associated with the addressed partition is active and does not match the write key register.

0E Read Time Stamp Instruction

The **Read Time Stamp** instruction reads the local real-time clock value stored in network memory at the address specified by the contents of the address reference register plus the unsigned contents of the instruction offset field. It then converts it into the global time format and places the result into the byte string immediately following the instruction. The instruction value field specifies the byte string length as 8 bytes.

The global time format consists of 8 bytes. The four most-significant bytes specify the time in seconds. The four least-significant bytes specify the binary fraction of a second. The value is in big endian format.

The instruction generates an error, rather than performing the read, if the specified string extends beyond the limit of the addressed partition, or if the read key associated with the addressed partition is active and does not match the read key register.

0F Write Time Stamp Instruction

The **Write Time Stamp** instruction writes the value of the network memory host real-time clock into the network memory at the address specified by the contents of the address reference register plus the unsigned contents of the instruction offset field.

To avoid an impact on performance, this instruction does not specify the format of the real-time clock value actually stored in network memory other than to limit it to no larger than 8 bytes. The network memory host is free to use its local real-time clock format and to store the value in either big endian or little endian format.

The instruction generates an error, rather than performing the write, if the specified string extends beyond the limit of the addressed partition, or if the write key associated with the addressed partition is active and does not match the write key register.

7. NETWORK IMPLEMENTATION CONCEPTS

This section presents the relationship of the network memory protocol implementation to the network layer and encapsulation concepts.

7.1 NETWORK LAYERS

Network software is typically built in layers. At the top is the socket layer which provides the user interface to the network through system calls. At the bottom is the network hardware driver. Between the top and bottom layers are software layers implementing one or more protocols such as the Internet Protocol (IP) and the Transmission Control Protocol (TCP). When the user writes a message to a socket, the message passes from the socket layer downward in sequence through the protocol layers until eventually reaching the driver that transmits it over the physical media. Conversely, when the driver for the physical media receives a message, the message passes from the driver upward in sequence through the protocol layers until eventually reaching the socket layer where it is made available for the user to read.

Network software is partitioned into layers so that various combinations of protocols at the different layers, including those for the physical media, can be provided by selecting from among the possible implementations at each layer. One can select, for example, either the TCP protocol or the Express Transport Protocol (XTP) above the IP protocol, and either the Ethernet protocol or the Fiber Distributed Data Interface (FDDI) protocol below the IP protocol. The layered partitioning allows the IP protocol processing to reside within a single implementation regardless of the particular protocols above or below it.

In actual practice, however, the layers are not entirely independent of one another. When a protocol passes a received message to a protocol immediately above it, for example, the lower protocol must determine which upper protocol to select. A field in the header processed by the lower protocol specifies the desired upper protocol. The Ethernet protocol uses the "Ether type" field in the thirteenth and fourteenth bytes of the Ethernet header. The IP protocol uses the "protocol" field in the tenth byte of the IP header. Implementing a new upper protocol requires a modification to all the existing lower protocols that interface to it so that the lower protocols can use their respective protocol selection fields to identify the new upper protocol.

Since the network memory protocol is intended to support high-speed, real-time database applications, it is especially important to minimize the round trip delay through the various network layers. An obvious way to do this is to eliminate some of the layers from the round-trip path. For example, the network memory protocol layer can be placed immediately above the Ethernet protocol layer. This has the advantage of very fast server response time. It has the disadvantage that the clients must deal with absolute media access control addresses and limit the length of their messages. Alternatively, the network memory protocol layer can be placed immediately above the IP protocol layer. The IP layer provides more addressing flexibility and overcomes the limit on message length at the cost of additional delay in the round-trip path.

A network memory host can support both approaches simultaneously. It is, therefore, useful to separate the portion of the network memory protocol software that implements the instruction set from the portions that interface the instruction set processing to the other network protocols at various layers.

7.2 ENCAPSULATION

Each protocol layer adds protocol-specific information to messages before passing them downward to the next-lower protocol layer. The additional information may include such items as protocol-specific source and destination addresses, control parameters, and check sums. Conversely, each layer removes information specific to itself from a message before passing the message upward to the next-higher layer. The process of enclosing a message within a protocol-specific conceptual envelope of additional information is called **encapsulation.**

The encapsulation concept is detrimental to a network memory protocol server layer placed immediately above an Ethernet layer. When the Ethernet layer receives a message, it strips the Ethernet header before passing the message to the layer above it. The Ethernet header contains the media access control source address of the client. The network memory protocol layer may construct a reply message. But if the Ethernet layer passes the network memory protocol layer a client message without the media access control source address, there is no way for the network memory protocol layer to inform the Ethernet layer of where to send the reply.

To solve the problem, the Ethernet layer implementation must be modified so that when it detects that a received message is to be passed to the network memory protocol layer, it does not strip the Ethernet header from the message before passing it.

8. NETWORK MEMORY PROTOCOL SERVER UNIX INTERNALS

This section describes issues related to the internal implementation of the network memory protocol server within the FreeBSD UNIX kernel. The availability of both the source code and textbooks describing the source code in detail led to the choice of the FreeBSD.

8.1 NETWORK MEMORY PROTOCOL INTERFACE TO THE ETHERNET PROTOCOL

Figure 7 shows the principal data paths among the FreeBSD UNIX kernel routines for the network memory protocol layer placed immediately above the Ethernet protocol layer.

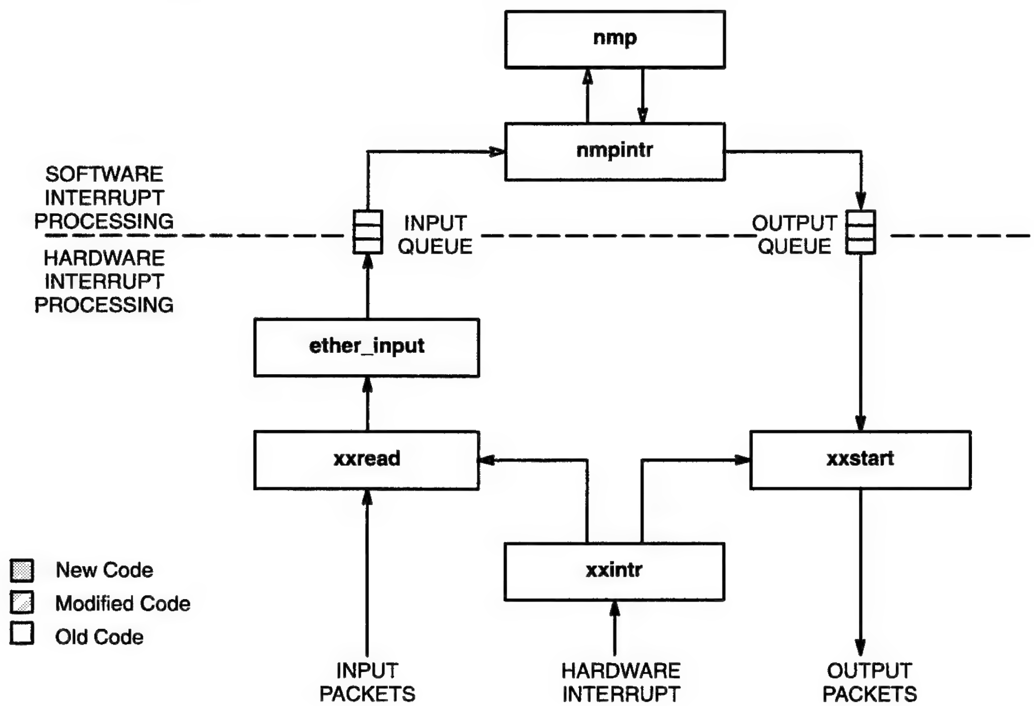

Figure 7. FreeBSD UNIX network memory protocol server implementation to the Ethernet protocol.

The routines whose names begin with the two letters "xx" are part of the Ethernet hardware driver code. Their implementation depends upon the particular hardware interface model installed. Each implementation of these routines has a name unique to the associated hardware model. The "xx" shown in the figure is symbolic for the unique portion of these names.

When an Ethernet packet arrives at the hardware interface, the hardware loads the packet into an input buffer and generates an interrupt. The `xxintr` routine receives all interrupts from the hardware interface. When it identifies the interrupt as indicating that a packet is now available in an input buffer, it calls the `xxread` routine to process the packet.

The `xxread` routine reformats each received packet as a chain of one or more mbuf's. An **mbuf** is a small chunk of kernel memory designed to accommodate the expansion and contraction of network packets due to encapsulation. The `xxread` routine calls a kernel version of the `malloc` routine to allocate mbuf's for the received packet.

The xxread routine calls the ether_input routine. The ether_input routine source code is in the file /usr/src/sys/net/if_ethersubr.c. The xxread routine passes three parameters to the ether_input routine.

The first parameter is a pointer to a structure called the ifnet structure. There is one ifnet structure for each network hardware interface. The ifnet structure maintains information unique to the associated interface.

The second and third parameters are pointers to the Ethernet header and the Ethernet data of the packet, respectively. Typically, these pointers point to bytes within the data area of the same mbuf such that the Ethernet data immediately follows the Ethernet header, as it does in the network media packet. The ether_input routine, however, does not require this. The xxread routine, therefore, is viewed as "stripping" the Ethernet header encapsulation from the packet before passing it to the ether_input routine.

As described in the previous section, the network memory protocol needs to keep the Ethernet header in order to obtain the Ethernet return address in case the received message generates a reply message. Thus, the ether_input routine must reattach the Ethernet header to the received messages. If the Ethernet header is already located in the proper place within the mbuf data area, this is simply a matter of moving the mbuf data pointer so that it points to the beginning of the Ethernet header rather than to the beginning of the Ethernet data.

The ether_input routine sorts the received mbuf chains into input queues based upon the content of the ether_type field in the Ethernet header. There is one input queue for each supported next higher protocol. The ether_input routine discards packets containing errors or requesting unrecognized next higher protocols. The standard ether_input routine must be modified to support the network memory protocol type field and input queue.

All the input processing to this point executes in response to an interrupt from the Ethernet hardware interface. Once all the pending packets reach their input queues, the ether_input routine triggers a software interrupt and exits. The software interrupt priority is lower than the hardware interrupt priority. The queue is necessary since several packets may be received at the high-priority hardware interrupt level before the low-priority software interrupt processing has an opportunity to execute.

The ether_input routine informs the kernel software interrupt handler of the next higher protocol input queues that contain pending mbuf chains through a global variable called the software interrupt word. Each input queue is assigned a unique bit position in the software interrupt word. The ether_input routine sets the associated bit when it enqueues an mbuf chain into an input queue. The software interrupt handler calls the routines for the particular next higher protocols based upon the bits that are set.

Network memory protocol message processing occurs at the kernel software interrupt level. It is partitioned into two routines, the nmpintr routine and the nmp routine.

If the software interrupt word bit assigned to the network memory protocol input queue is set, the kernel software interrupt handler calls the nmpintr routine. The nmpintr routine contains a loop that sequentially dequeues and processes each pending message in the input queue. It performs the network memory protocol message processing that is unique to the Ethernet. This consists of converting the received Ethernet header into a new Ethernet header for the reply message.

The nmpintr routine copies the media access control destination address for the reply message Ethernet header from the media access control source address of the request message Ethernet header. This will return the reply message back to the source of the original request message.

The header of the first mbuf of the message mbuf chain contains flags, initialized by the xxread routine, indicating the destination address type. For unicast messages, the nmpintr routine copies the reply message source address from the request message destination address. This does not work for multicast and broadcast messages because the request message destination address is not unique. These require the nmpintr routine to obtain the reply message source address from the network hardware interface ifnet structure. The mbuf provides the pointer to the ifnet structure. The xxread routine initialized the pointer, m_pkthdr.rcvif, by calling the m_devget function.

The nmpintr routine calls the nmp subroutine. The nmp subroutine performs the network memory protocol message processing that is common regardless of the network layers below the network memory protocol layer. It implements the network memory protocol instruction set. It receives the mbuf chain containing the received network memory protocol message and returns an mbuf chain, if appropriate, containing the generated reply message. It returns a null pointer if there is no reply message.

Although the nmp subroutine has no interest in the Ethernet header in the first mbuf of the mbuf chain, it must not discard the Ethernet header if the message generates a reply. The nmpintr routine that called the nmp subroutine needs the Ethernet header for the reply return address. The nmpintr routine, therefore, passes two parameters to the nmp subroutine, a pointer to the mbuf chain and an offset from the start of the data of the first mbuf of the chain where the nmp subroutine should begin processing. The nmp subroutine is expected to preserve any data between the beginning of the mbuf data and the beginning of the data it processes. Beyond this restriction, the nmp subroutine is free to allocate, modify, or free mbuf's in the chain, as necessary, to implement the network memory protocol instructions specified by the message.

If the request message does not generate a reply message, the nmp subroutine discards all the message mbuf's after processing them and returns a null pointer. The nmpintr routine, upon seeing the null pointer, proceeds to the next message mbuf chain in the input queue, and if there are none, exits.

If the request message does generate a reply message, the nmpintr routine enqueues the mbuf chain provided by the nmp subroutine into the Ethernet output queue. It then checks to see whether the hardware is currently transmitting messages from the output queue. If the hardware is transmitting messages, then the message added to the queue will be transmitted, in turn, without further intervention. If the hardware is not currently transmitting messages, then it must be explicitly started. The nmpintr routine informs the network interface driver to begin transmitting messages on the output queue by calling the xxstart routine. Finally, the nmpintr routine checks for remaining messages in the input queue and, if there are none, exits.

8.2 NETWORK MEMORY PROTOCOL INTERFACE TO THE INTERNET PROTOCOL

Figure 8 shows the principal data paths among the FreeBSD UNIX kernel routines for the network memory protocol layer placed immediately above the IP layer.

The network interface hardware loads the received packet into an input buffer and generates an interrupt. The xxintr routine identifies the hardware interrupt and calls the xxread routine. The xxread routine reformats the packet as an mbuf chain and calls the ether_input routine. The

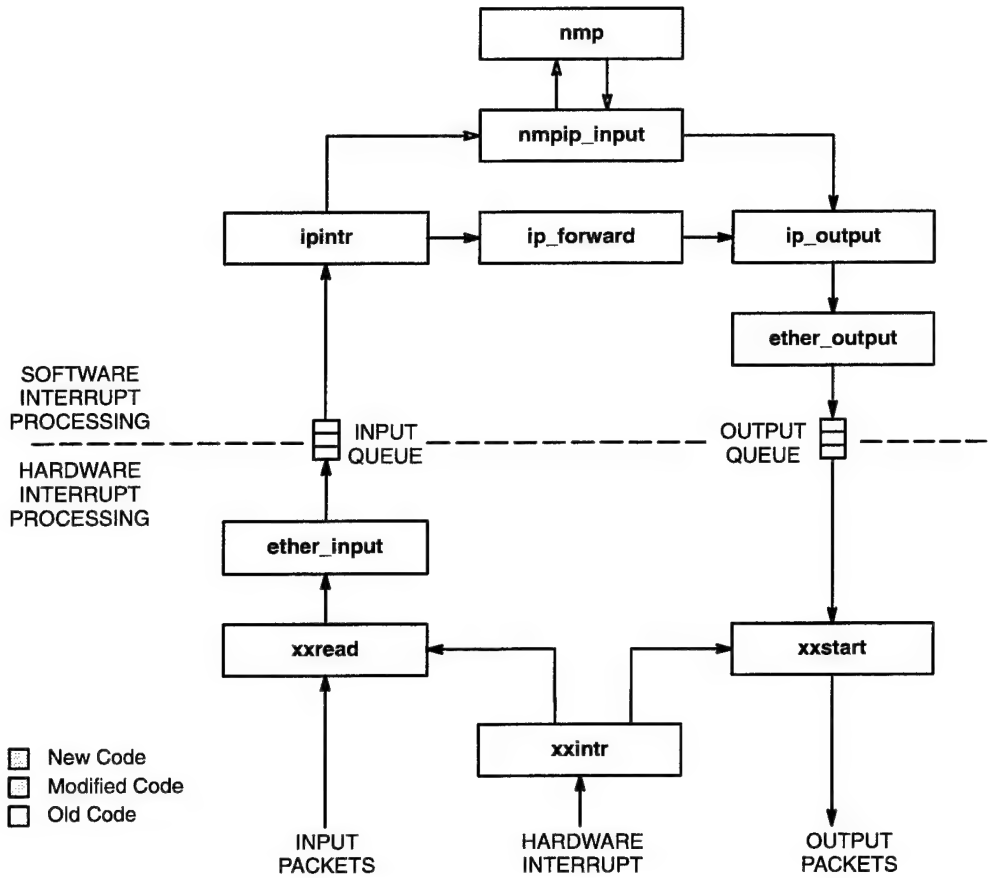

Figure 8. FreeBSD UNIX network memory protocol server implementation to the Internet protocol.

`ether_input` routine enqueues the mbuf chain into the input queue for the protocol specified by the Ethernet header `ether_type` field, sets the respective bit in the software interrupt word, and generates the software interrupt. IP protocol mbuf chains, including those encapsulating the network memory protocol, go into the input queue for the IP protocol.

The data content of an IP protocol mbuf chain is called a **datagram**. The encapsulated data of the datagram is called its **message**.

If the IP protocol bit in the software interrupt word is set, the software interrupt handler calls the `ipintr` routine. The `ipintr` routine source code is in the file `/usr/src/sys/netinet/ip_input.c`. The `ipintr` routine contains a loop that sequentially dequeues and processes each pending datagram mbuf chain in the input queue. The `ipintr` routine is a large and complex routine implementing datagram verification, option processing, forwarding, and reassembly, as well as passing the datagram mbuf chain to the next higher protocol layer. The following discussion only considers the IP protocol layer interface to the network memory protocol layer.

The protocol field in the IP header contains a number identifying the next higher protocol intended to receive the datagram. The `ipintr` routine uses this number as an index into an array called the `inetsw` array. The `inetsw` array converts the protocol number into a new index, this time to an entry within a table called the `protosw` table. The `pr_input` field of the selected entry contains

the address of the input routine for the next higher protocol associated with that entry. The ipintr routine calls that input routine to pass the datagram mbuf chain to the next higher protocol layer.

The protosw table entries also contain a field for the protocol number. One could locate an entry for a particular protocol by sequentially searching the table for the entry with the desired value in that field. The purpose of the inetsw array is to replace the slow search with a fast direct mapping. The kernel network software generates the inetsw table during initialization based on the protocol numbers specified within the protosw table entries.

If the IP header protocol field indicates that the datagram is for the network memory protocol, the ipintr routine calls the nmpip_input routine. The nmpip_input routine performs the network memory protocol processing that is unique to the IP protocol. It converts the received IP header into one suitable for the reply back to the client. The nmpip_input routine calls the nmp subroutine, the same one as described in the previous section, to perform the processing that is common regardless of the network layer that is below the network memory protocol layer. It implements the network memory protocol instruction set.

When the ipintr routine passes the datagram mbuf chain to the nmpip_input routine, it still contains the IP header before the message data. The header is normally 20 bytes long, but it could be longer if there are IP options. The nmpip_input routine passes the header length along with a pointer to the mbuf chain to the nmp subroutine. The nmp subroutine has no interest in the header, but it preserves the header so that it is available to the nmpip_input routine when the nmp subroutine returns.

Sending a network memory protocol reply message back to the client is considerably more complex immediately above the IP layer than it is immediately above the Ethernet layer. Since the Internet Control Message Protocol (ICMP) also sends reply messages immediately above the IP layer, it provides a coding templet. Its icmp_reflect routine returns datagrams containing reply messages. The source code is in the file /usr/src/sys/netinet/ip_icmp.c.

The IP protocol does not modify its header source and destination Internet address fields as a datagram is forwarded through routers. The nmpip_input routine uses the request datagram Internet source address for the reply datagram Internet destination address. Thus, the initial source of the network memory protocol message becomes the final destination of the network memory protocol reply message.

If the request datagram Internet destination address is a unicast address, the nmpip_input routine uses the request datagram Internet destination address as the reply datagram Internet source address. If it is not a unicast address, it uses the Internet address of the network hardware interface that received the datagram as the reply datagram Internet source address.

The nmpip_input routine sends the reply datagram by calling the ip_output routine. The ip_output routine has several input parameters. The first parameter is a pointer to the reply datagram mbuf chain. The second parameter, intended for a pointer to an mbuf chain of options, is set to a null pointer since options are not currently implemented. The third parameter, intended for a pointer to a route cache of destination addresses, is set to a null table since the nmpip_input routine does not maintain such a cache. The fourth parameter, used for flags, is set to zero. These flags include the IP_ALLOWBROADCAST flag. Setting it to zero requests the ip_output routine to discard any reply datagrams having broadcast destination Internet addresses. The only way that a reply datagram

could have a broadcast Internet destination address is if the request datagram that generated the reply had a broadcast Internet source address, and that would be illegal.

9. NETWORK MEMORY PROTOCOL UNIX KERNEL MODIFICATIONS

The procedures for installing a new driver and for compiling a new kernel are covered in operating system manuals. The procedure for installing a new local area network protocol is seldom presented. This section describes the modifications to FreeBSD UNIX version 2.2.1 required to install the network memory protocol on an Intel 386 or higher personal computer.

9.1 NETWORK MEMORY PROTOCOL INTERFACE TO THE ETHERNET PROTOCOL

Add the following line to file `/usr/src/sys/netinet/if_ether.h` to define the value of the Ethernet type field used for the network memory protocol.

```
#define ETHERTYPE_NMP 0x9000    /* network memory protocol */
```

The value of 9000 hexadecimal was chosen arbitrarily. It must be unique.

Add the following line to file `/usr/src/sys/net/netisr.h` to define a bit in the software interrupt word indicating that there is data in the network memory protocol input queue.

```
#define NETISR_NMP 28           /* network memory protocol */
```

The bit position, in this case 28, must be unique.

Add the following line to the file used to specify the configuration which was edited from the file `/usr/src/sys/i386/conf/GENERIC`.

```
options     ETHERNMP    #Ethernet network memory protocol
options     NMP         #network memory protocol
```

This defines the `ETHERNMP` macro, used to include network memory protocol code at the Ethernet layer in the compile of routines where it is an option, and the `NMP` macro, used to include the common `nmp.c` code.

Add the following two lines to the file `/usr/src/sys/conf/files` to include the two listed files in the compile of the kernel.

```
netnm/nmp_ether.c       optional ethernmp
netnm/nmp.c             optional nmp
```

The `nmp_ether.c` code implements the Ethernet input queue software interrupt processing. It calls the `nmp.c` code, which implements the network memory protocol instruction set.

Add the following code to the file `/usr/src/sys/net/if_ethersubr.c` near the top where there are similar declarations.

```
#ifdef ETHERNMP
extern struct ifqueue nmpintrq;
void nmp_init();
#endif
```

These lines declare the input queue for the network memory protocol software interrupt and the subroutine used to initialize the network memory protocol variables.

Also add the following code to the file /usr/src/sys/net/if_ethersubr.c, within the routine ether_input, as an ether_type case to the switch statement.

```
#ifdef ETHERNMP
        case ETHERTYPE_NMP:
                M_PREPEND(m, ETHER_HDR_LEN, M_DONTWAIT);
                if((caddr_t)eh != (caddr_t)m->m_data) {
                        bcopy((caddr_t)eh, (caddr_t)m->m_data,
                                 ETHER_HDR_LEN);
                }
                schednetisr(NETISR_NMP);
                inq = &nmpintrq;
                break;
#endif
```

The M_PREPEND macro makes space available immediately before the existing data in the first mbuf. This space will hold the Ethernet header so that the return address is available in case the message generates a reply. The if statement checks whether the header is already present in the space and, if not, copies it into the space. The schednetisr macro takes care of the details of setting the bit dedicated to the network memory protocol in the software interrupt word. The assignment statement before the break statement identifies the input queue where the mbuf chain is enqueued for the software interrupt.

Add the following code to the source file /usr/src/sys/net/if_ethersubr.c, within the routine ether_ifattach, at the end of the routine.

```
#ifdef ETHERNMP
nmp_init();
#endif
```

The nmp_init routine initializes the length of the network memory protocol software interrupt input queue. The source is in the file /usr/src/sys/netnm/nmp_ether.c.

```
struct ifqueue nmpintrq;
int nmpqmaxlen = IFQ_MAXLEN;
void
nmp_init(void)
{
        nmpintrq.ifq_maxlen = nmpqmaxlen;
}
```

The single statement of the nmp_init routine could have been placed in the ether_ifattach routine rather than in a separate routine. It is placed in a separate subroutine for modularity.

The source file /usr/src/sys/netnm/nmp_ether.c also contains a macro executed during kernel initialization to link the nmpintr routine and the associated software interrupt word bit position NETISR_NMP to the software interrupt handler.

```
NETISR_SET(NETISR_NMP, nmpintr);
```

The remainder of the source file contains the nmpintr routine.

9.2 NETWORK MEMORY PROTOCOL INTERFACE TO THE INTERNET PROTOCOL

Add the following line to file `/usr/src/sys/netinet/in.h` to define the protocol number for the network memory protocol.

```
#define IPPROTO_NMP  200   /* network memory protoco */
```

The protocol number is the number that appears in the tenth byte of the IP header. The value 200 decimal was chosen arbitrarily. It must be unique.

Add the following line to the file used to specify the configuration which was edited from the file `/usr/src/sys/i386/conf/GENERIC`.

```
options    IPNMP#IP network memory protocol
options    NMP         #network memory protocol
```

The second line is not added if already present. This defines the `IPNMP` macro, used to include network memory protocol code at the IP layer in the compile of routines where it is an option, and the `NMP` macro, used to include the common `nmp.c` code.

Add the following two lines to the file `/usr/src/sys/conf/files` to include the two listed files in the compile of the kernel.

```
netnm/nmp_ip.c          optional ipnmp
netnm/nmp.c             optional nmp
```

The second line is not added if already present. The `nmp_ip.c` code implements the network memory protocol interface to the IP layer. It calls the `nmp.c` code, which implements the network memory protocol instruction set.

Add the following code to the file `/usr/src/sys/netinet/in_proto.c` near the top where there are similar declarations.

```
#ifdef IPNMP
extern void nmpip_input __P((struct mbuf *, int))
#endif
```

The `__P` macro define the C language types of the two input variables of the `nmpip_input` routine.

Also add the following code to the file `/usr/src/sys/netinet/in_proto.c` as an entry within the `protosw` table structure.

```
#ifdef IPNMP
{     SOCK_RAW, &inetdomain,    IPPROTO_NMP,
      PR_ATOMIC|PR_ADDR,
      nmpip_input,    0,        0,          rip_ctloutput,
      rip_usrreq,
      0,              0,        0,          0,
},
#endif
```

This table entry links the protocol number `IPPROTO_NMP` to the `nmpip_input` routine.

10. NETWORK MEMORY PROTOCOL SERVER TESTING

This section describes the computer system used to test network memory protocol server implementations and to measure their performance.

10.1 TEST SYSTEM

Figure 9 show the test system configuration for the network memory protocol server placed immediately above the Ethernet protocol layer.

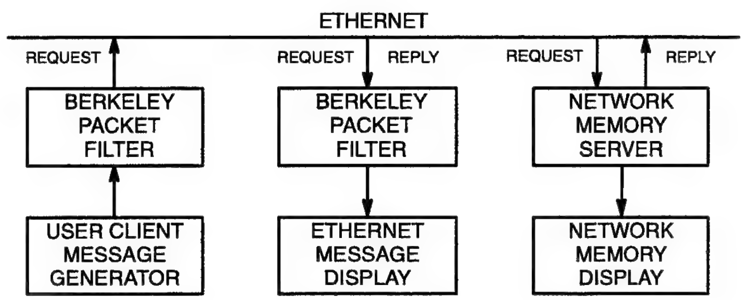

Figure 9. Network memory protocol performance test configuration.

The Berkeley Packet Filter is a UNIX driver permitting an application program to read and write raw Ethernet packets. The filter feature of the driver permits an application to write a logic formula specifying the field contents for the packets that the driver receives. The Berkeley Packet Filter also time-stamps each packet received.

The Berkeley Packet Filter is usually included in the FreeBSD source code product, but is not part of the default kernel configuration. It can be compiled into the kernel by adding the following line to the configuration file derived from the file `/usr/src/sys/i386/conf/GENERIC`.

```
#pseudo-device  bpfilter   4     #Berkeley packet filter
```

The number, in this case 4, specifies the number of simultaneous opens permitted by the driver.

As shown in figure 9, the test system consists of three processors.

The left processor generates network memory protocol sensor client request messages. It writes the messages to the Ethernet through the Berkeley Packet Filter driver. It is used to generate both valid messages and messages deliberately containing protocol errors to test the network memory protocol server functional implementation.

The right processor implements the network memory protocol server under test. In addition to the server, it implements an application program with access to the network memory. This program is used to verify that network memory protocol server writes to the network memory were performed properly.

The middle processor is used to monitor the network memory protocol message traffic, both requests and replies, that passes over the Ethernet interconnecting the three processors. The test system avoids introducing measurement delays by isolating the monitoring function in a separate processor. The Berkeley Packet Filter driver selects the packets of interest. The display program is used to

verify network addressing and the operations performed on protocol fields. The display program also receives the time stamps generated by the Berkeley Packet Filter driver. These time stamps are the basis for computing server performance.

The Berkeley Packet Filter driver obtains time-stamp values from the kernel `microtime` routine. The network memory protocol server uses the same routine in the code for its Write Time Stamp instruction. The routine provides a resolution in the neighborhood of 10 to 20 microseconds, depending upon the supporting processor hardware.

10.2 TEST RESULTS

Comparisons were made between simple database transactions performed by a network memory protocol server located immediately above the Ethernet layer versus the same transactions performed by a server interfaced to a User Datagram Protocol (UDP) socket. In all the cases tested, the network memory protocol server was faster. This is to be expected for the many reasons given in the previous section comparing the network memory approach to the socket approach. In fairness to the UDP protocol, however, it needs to be noted that the UDP performs services, including datagram routing, reassembly, and error-checking, which are not performed by the network memory protocol immediately above the Ethernet layer.

Server performance is computed as the average round-trip delay from when the client sends the request message under test to when the server sends the reply message back to the client. It is important to execute performance tests on a quiet local area network so that unrelated network traffic does not interfere with the measurements.

Tests were performed using a 50 MHz Intel 486 personal computer for the network memory host and a Pentium personal computer for the Ethernet monitor. The network memory protocol `nmp` routine was modified to work with user buffers rather than mbuf's for the servers employing sockets. The round-trip times for the simple transactions, such as addressing and then reading or writing a few bytes, were approximately 200 microseconds for the network memory protocol server, approximately 330 microseconds for the UDP socket server, and approximately 750 microseconds for the TCP socket server.

There is a slight performance gain of about 10 to 20 microseconds when network memory protocol client request messages are sent in rapid succession rather than separated in time. When messages arrive in rapid succession, they enter the input queue and the `nmpintr` routine processes them as a group in response to a single software interrupt. When they are separated in time, each enters the input queue and is processed in response to a separate software interrupt. The latter is less efficient due to the additional software interrupt context switching and `nmpintr` routine initializations. Clients can reduce network memory server software interrupts and, thus, enhance server performance, by grouping several transactions into the same local area network message.

11. SUMMARY OF CONCLUSIONS

The network memory protocol implements local area network access to high-throughput, real-time databases more efficiently than servers employing traditional sockets. The efficiency is gained by avoiding unnecessary context switches and data buffer copies. Since the network memory protocol provides random access to data, it is more flexible than socket approaches, which are inherently sequential. The socket approach, on the other hand, does a better job of activating tasks in response to associated message arrivals.

User-defined partitions and overlays can organize the network memory host user address space independently of the global network memory protocol address space. Partitioning can conserve the amount of kernel memory allocated to network memory. It can also map non-contiguous kernel address spaces for access to different types of memory, such as memory-mapped input/output hardware.

UNIX application software interfaces to the network memory through a network memory character device driver. The first `open` system call allocates the network memory within kernel memory and the last `close` system call frees it. The `ioctl` system call defines the partitions. The `mmap` system call maps the allocated network memory into the application user space. The application then has direct access to the network memory without further system calls.

The network memory protocol server, implemented within the kernel, may access the network memory records while an application task was in the process of accessing the same records. Coordination algorithms, which avoid system calls, have been developed to ensure atomic access to network memory data. The functions required by these algorithms, plus the need for real-time clock synchronization in real-time system, guided the design of the network memory protocol message format.

Network memory protocol messages are composed of instructions that together form a simple sequential program. The network memory protocol server can be viewed as a very simple processor that executes the program defined by the instructions forming the message. The instructions support both direct and pointer access to data, a few simple logic and arithmetic functions, time-stamping, and security keys.

Network memory protocol interfaces to both the Ethernet layer and the Internet Protocol layer have been presented. Immediately above the Ethernet layer, the network memory protocol server was found to execute approximately a third faster than an equivalent server using traditional sockets to the User Datagram Protocol.

This research has explored the feasibility and potential of the network memory protocol. Additional work will be necessary to convert the experimental software into a fully functional product ready for widespread distribution.

The research reported in this paper relies upon no special-purpose hardware. It integrates the network memory protocol into the set of existing network software. The simplicity of the network memory protocol instructions suggests that the network memory protocol server could be implemented in a hardware memory system with a local area network interface, rather than in software executing within a host processor. This would be an area for possible future research.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1997	Final: Oct 1996 to Sep 1997

4. TITLE AND SUBTITLE
NETWORK MEMORY PROTOCOL

5. FUNDING NUMBERS
PE: 0601152N
AN: DN306757
WU: ZU36

6. AUTHOR(S)
D. R. Wilcox

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Command, Control and Ocean Surveillance Center (NCCOSC)
RDT&E Division (NRaD)
San Diego, CA 92152–5001

8. PERFORMING ORGANIZATION REPORT NUMBER
TR 1757

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Space and Naval Warfare Systems Command
Washington, D.C. 20363–5001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report presents initial research into the design of a new computer system local area network transport layer protocol, designated the network memory protocol, which provides clients with direct access to data within the random access memories of servers. The server function executes within the operating system kernel without the use of sockets. The protocol is intended primarily for high-throughput, real-time database systems. The report presents the motivation for design decisions regarding address mapping, message formats, atomic coordination, time synchronization, and UNIX kernel implementation. Server performance results are compared with those for servers employing socket protocols.

14. SUBJECT TERMS
Mission Area: Command, Control, and Communications
network memory protocol — local area network transport layer protocol
real-time databases

15. NUMBER OF PAGES
45

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAME AS REPORT

21a. NAME OF RESPONSIBLE INDIVIDUAL	21b. TELEPHONE *(include Area Code)*	21c. OFFICE SYMBOL
D. R. Wilcox	(619) 553-5467 e-mail: wilcox@nosc.mil	Code D851

INITIAL DISTRIBUTION

Code D0012	Patent Counsel	(1)
Code D0271	Archive/Stock	(6)
Code D0274	Library	(2)
Code D027	M. E. Cathcart	(1)
Code D0271	D. Richter	(1)
Code D41	R. B. Volker	(1)
Code D4102	M. L. Crowley	(1)
Code D4102	H. Gold	(1)
Code D4103	L. J. Core	(1)
Code D411	M. Vineberg	(1)
Code D4122	M. W. Morgan	(1)
Code D4122	P. N. Pham	(1)
Code D4122	D. R. Wilcox	(30)
Code D421	B. Bonwit	(1)
Code D431	R. Millen	(1)
Code D431	D. G. Mudd	(1)
Code D432	R. Heidecker	(1)
Code D4324	S. M. Luby	(1)
Code D441	D. Gomez	(1)
Code D441	T. Tiernan	(1)
Code D44212	B. Laughlin	(1)
Code D44212	R. E. Luna	(1)
Code D44212	L. A. Wise	(1)
Code D44214	D. Lasniewski	(1)
Code D45	K. M. Register	(1)

Defense Technical Information Center
Fort Belvoir, VA 22060–6218 (4)

Center for Naval Analyses
Alexandria, VA 22302–0268

Navy Acquisition, Research and Development
Information Center (NARDIC)
Arlington, VA 22244–5114

GIDEP Operations Center
Corona, CA 91718–8000

SPAWARSYSCEN Liaison Office
Arlington, VA 22202–4804